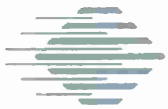
Technical Report

CMU/SEI-87-TR-18
ESD-TR-87-126



Carnegie-Mellon University
Software Engineering Institute

The Use of Representation Clauses and Implementation-Dependent Features in Ada:

IIB. Experimental Procedures

B. Craig Meyers
Andrea L. Cappellini

July 1987

Technical Report
CMU/SEI-87-TR-18
ESD/TR-87- 125
July 1987

The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIB. Experimental Procedures

B. Craig Meyers
Andrea L. Cappellini

Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Daniel Burton
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIB. Experimental Procedures

Abstract: This report is one in a series dealing with the use of representation clauses and implementation-dependent features in Ada. The purpose of this report is to discuss detailed experimental procedures to assess compiler support. It is readily acknowledged that the domain of possible experimentation is large. To facilitate the experimentation, a methodology is proposed that relies on program generators and automated analysis tools. An example of the methodology is presented in some detail.

1. Introduction

The Ada language was developed as a general-purpose language with specific application to mission-critical systems for the Department of Defense (DoD). The language is now mandated for use in real-time, mission-critical systems, as specified in reference [1].

In spite of the attempt to define Ada as a general-purpose language, a need to support implementation-specific functionality was recognized. This amounted to the language providing for a coupling to the underlying architecture. Much of this support is defined in terms of representation clauses and implementation-dependent features which are discussed in Chapter 13 of the *Reference Manual for the Ada Programming Language* [2].

Representation clauses and other machine-dependent features provide the capability to interact between an application program and the machine. For example, pragma PACK may be specified to minimize storage allocation for array and record types. There are also representation clauses for specifying the amount of storage to be used for a particular type, specifying the values associated with enumerated data types, and defining the precise storage layout of data within a record structure. Furthermore, there are representation clauses that allow a program to access a specific address and specify information concerning tasking. The functionality provided by representation clauses and implementation-dependent features may be needed in existing systems. However, the amount of support provided for a particular representation clause is largely left to the compiler, and is, therefore, likely to be implementation-dependent. Also, the use of representation clauses may affect the portability of the code.

This report is one of a series dealing with the use of representation clauses and implementation-dependent features in Ada. The first volume in the series, reference [3], provides an overview of the use of these clauses and features. That document draws on examples from mission-critical systems to illustrate the use of representation clauses and implementation-dependent features. A set of questions was developed that is believed relevant to the consideration of the use of representation clauses and implementation-dependent features [4]. The questions defined in reference [4] have been examined in some detail from a qualitative perspective for two Ada compilers [5], [6].

In view of the fact that representation clauses and implementation-dependent features are deemed necessary by many developers and are implementation specific, there is a clear need to be able to quantitatively assess the support of these features provided by particular compilers. It is the purpose of the present report to fulfill that need. Thus, the principal concern here is with the development of experimental procedures. This leads to the formulation of a larger context for which experimentation may be conducted. The application of a methodology such as that developed here may be useful in the evaluation of an Ada compiler with regard to the support provided for machine-dependent features.

We are concerned here with quantitative aspects of compiler assessment. A broad interpretation of quantitative measures is assumed. That is, not only are we concerned with traditional performance measures such as execution time and storage allocation, but we are also concerned with the manner in which a particular feature is implemented. This may be obtained by examining the assembler code produced by a compiler.

This report is organized in the following manner: Chapter 2 presents a brief discussion concerning the purpose of performing experimentation and the associated scope. The discussion of experimental context ranges from case studies to detailed experiments. Chapter 3 provides a discussion of the detailed design of experiments with emphasis on evaluating a particular compiler. A discussion of a methodology for the conduct of experiments is presented in Chapter 4. A summary of the document is given in Chapter 5.

This report has been prepared by the Ada Embedded Systems Testbed project at the Software Engineering Institute (SEI). The SEI is a federally funded research and development center (FFRDC) sponsored by the Department of Defense (DoD) and established and operated by Carnegie Mellon University. This report is based on work performed by the authors while they were on sabbatical leave at the SEI.

2. Purpose and Scope

The range of investigations possible for any particular feature(s) supported by a compiler is extensive. To set the stage for what follows, we provide some initial comments concerning the purpose and scope of investigations.

2.1. Purpose

Some discussion of the goals of performing experiments is warranted. In the general context, recall that the experiments are designed to measure some characteristic features of the manner in which a compiler supports representation clauses and implementation-dependent features. We characterize three of the goals of experimentation in terms of the following classes:

1. To assess a particular compiler for use in an independent manner.
2. To assess a particular compiler used in relation to some other compiler.
3. To compare the manner in which a compiler implements some function in relation to an alternate, user-chosen method of implementing the same functionality.

The conduct of detailed assessments to meet the first two goals listed above may be useful in compiler development and compiler selection, respectively. The latter case may be motivated by questions of efficiency. Thus, if some particular functionality is required, an assessment may indicate that an alternative method may be warranted.

A considerable amount of information may be available for each of the goals listed above. These may be broadly characterized as:

- performance measures
- qualitative measures

In the first case, we are concerned with traditional assessment characteristics, such as code size and execution time. In the second case, we are concerned with extraction of information concerning the mechanism of code generation. This may include an examination of instruction set usage or analysis of branch statements generated by the compiler. It is this second category – obtaining information about *how* a feature is implemented – that provides considerable challenge. Additionally, it is this type of information that may be of special interest.

2.2. Scope

There are a variety of ways an investigation may be conducted and a wide range of possible investigations. In particular, we may consider the following as examples tending to define the boundaries of possible investigations:

1. case studies
2. detailed experiments

A case study approach is often taken for a variety of reasons. First, it may be representative of the

problem under consideration. That is, the case study example may be representative of a general class of problems. A second reason for performing a case study is that it is considerably less labor-intensive than detailed experimentation.

In contrast to a case study, detailed experiments are often confined to a more limited domain. This is due, in part, to the effort required to appropriately formulate and conduct experiments.

It may be useful to provide some comment concerning the use of "benchmarks" and how they relate to the above two approaches. We include benchmarks as a limited form of detailed experiment. Here, however, a distinction exits between a benchmark test and a detailed experiment. That is, typically a benchmark test is used to obtain performance information about the compiler such as characteristic time and storage values. We choose not to limit the domain of information one would like to obtain from the performance of a detailed experiment. As indicated above, a detailed experiment may be concerned with performance measures. It may also be concerned with the recording and analysis of information about *how* the compiler performs some function.

Both a case study approach and the use of detailed experiments have respective strengths and weaknesses. We are, in general, opposed to an *arbitrary* case study approach. This is based principally on two factors. First, it is difficult, if at all possible, to justify the choice of parameters used in the case study. Second, it is not clear to what degree the results of a case study may be generalized. That is, it is typically assumed that the generalization has been done in the *beginning* of the experiment. In contrast, a detailed experiment is capable of some generalization in a formal sense. Such generalizations are typically developed *after* the experiment has been completed.

What one would perhaps desire is the development of a methodology that may serve both types of investigations noted above. Specifically, the methodology would include a framework that could support the use of case study approaches. At the same time, the methodology must be able to account for the desire, indeed the need, to perform detailed experimentation. Should such a formalism be founded on a case study basis, or should it be developed from the perspective of a detailed experiment?

We believe that the development of a consistent approach to assessment must, of necessity, be formulated from consideration of a detailed experimental procedure. The reason for this has already been stated above. In particular, it is often difficult to justify the choice for the case study or to place bounds on the domain to which it applies. On the other hand, it is clear that the application of detailed experimentation may involve considerable labor.

The methodology to be developed here, being based on a detailed approach, must address the issue of the resource intensiveness. This we propose to do. Specifically, we propose a novel technique that relies heavily on program generators and automated analysis tools. When used in other domains, this approach has, indeed, removed questions dealing with the amount of labor involved. Additionally, detailed experimentation may provide justification for the use of case studies, where appropriate. That this is possible is indicated with the methodology to be discussed, particularly when the set of experiments is viewed in a formal sense.

3. Experimental Design

In the previous chapter, we discussed various approaches to the problem of securing information about an implementation of a particular aspect of representation clauses and implementation-dependent features. The available investigations ranged from ad hoc case studies to detailed experimentation. The purpose of this chapter is to expand on the discussion of detailed experimentation.

3.1. Experimental Design Formalism

This chapter establishes a general framework for the design of a detailed experiment. The goal of the experiment is only loosely stated, and is, of necessity, a function of the particular compiler being evaluated. The establishment of a framework for the design of experiments is, in itself, tantamount to the development of a methodology. However, an equally relevant methodology deals with the application of the results, or guidelines, from the design of an experiment. That is, given that some method exists to define a detailed experiment, how is the experiment applied in a particular case? This question refers to a more general methodology for the conduct and assessment of experiments and is discussed in detail in the following chapter.

The formalism to be adopted for the delineation of a detailed experiment, for a given aspect of representation clauses and implementation-dependent features, is outlined in the following:

1. Identify a group of sets, to be called "principal" sets, that define the basic parameters of the experiment.
2. Define elements of the principal sets that are the particular values applicable to the experiment. The elements of a principal set define the domain of an experiment.
3. Identify the measurements that are relevant for the experiment and those qualitative issues that may be addressed.

The above procedures are applied in the following subsections to define a set of general experiments intended to deal with representation clauses and implementation-dependent features. Hence, in the following, experiments are described that are applicable to a class of representation clauses and implementation-dependent features. The "dimensionality" of an experiment is a function of the principal sets and their respective domains. In general, the dimensionality may be quite large. It is precisely for this reason that novel approaches are required to handle the large amount of information available. These issues are more appropriately discussed within the framework of an experimental methodology and are addressed in Chapter 4.

The classes of experiments to be discussed below address the following:

- pragma PACK
- length clauses
- enumeration representation clauses
- record representation clauses
- address clauses
- data conversion and assignment

- representation attributes

Some overlap might exist in the experiments defined below, depending on the enumeration procedures used.

3.2. Experimentation for Pragma PACK

Ada provides a pragma, PACK, which will cause the storage allocated by the compiler for arrays and records to be minimized. It is to be emphasized that the storage allocation is minimized. This does not imply that data objects are mapped contiguously onto storage, however. For this experiment, the principal sets and their associated members are:

1. *Unit type*, which has two members, namely, array and record.
2. *Compiler state*, whose members represent no optimization, optimization of storage, and optimization of time.
3. *Data types*, whose members include enumeration, character, integer, fixed-point, and floating-point types.
4. *Size*, whose members represent the amount of storage (in bits) allocated for a member of the set data types. Thus, this set may be specified in terms of the integers in the range 1, 2, ..., P_MAX_SIZE where P_MAX_SIZE denotes the maximum size to be considered for a particular data type.
5. *Order*, whose members are themselves sets that specify the ordering of the possible combinations of elements from the principal set of data types.
6. *Number of elements*, whose members represent either the number of elements in an array or components in a record. This set is the integers within 1 .. P_MAX_NUM where P_MAX_NUM denotes the maximum number of elements or components to be considered.

The measurables for this experiment are the compile time and the size of the code that is generated. It is assumed that an assessment of the storage allocation mechanism will be performed.

The set *unit type* is identified by the Ada language constraint that only array and record types may be packed. The compiler state has been included; although the effect of optimization is typically of greatest interest, it may also be useful to have some measure of the results for a nonoptimizing state of the compiler. The principal set of *data types* has elements which represent the basic data types to be considered. Following this, the size of storage allocated for a given type is listed.

The principal set *order* specifies the ordering of the basic data types. This principal set has elements that are also sets, to account for the various combinations of the basic data types. Hence, a possible member of this set would be a set that contains the elements enumeration, enumeration, and integer. Another member of the ordering set would be a set containing the elements enumeration, integer, and enumeration. Note the implicit ordering required here. This amounts to recognizing that the effect of storage allocation in the presence of pragma PACK may depend on the ordering of the basic elements. The last principal set specifies the number of array elements or record components to be considered. It is the cardinality of this set that governs the "dimensionality" of the experiment.

The performance of detailed experimental assessments implies a considerable number of experi-

ments must be conducted. For the cases under consideration, this means creating, compiling, and collecting information for an Ada procedure. Let us consider an example of a typical experiment for pragma PACK. Thus, we consider an experiment illustrating principal sets as follows:

- unit type: array
- compiler state: optimize storage
- data type: integer
- size: 16 bits
- order: not applicable
- number of elements: 20

An example of an Ada procedure to assess pragma PACK for the values specified above is shown in Figure 1. A complete assessment of pragma PACK would require assessing many procedures similar to that illustrated in Figure 1. A procedure Initialize_Array is invoked to initialize values of the data structure. This once again serves to illustrate the need for automated procedures within a methodological framework.

Figure 1
Ada Procedure To Assess Pragma PACK

```
procedure Pack_Example is

  pragma Optimize(Space);

  type Array_Element_Type is range 0 .. 32000;
  for Array_Element_Type'SIZE use 16;

  type Array_Type is array(1 .. 20) of Array_Element_Type;
  pragma PACK(Array_Type);

  Example_Array : Array_Type;

begin

  Initialize_Array (Example_Array);

end Pack_Example;
```

3.3. Experimentation for Length Clauses

The length clause specifies a maximum amount of storage associated with objects of a particular type. The following are the principal sets and their members for experimentation of length clauses:

1. *Compiler state*, whose members are no optimization, optimization of space (storage), and optimization of time.
2. *Attribute class*, whose members are 'Size, 'Storage_Size, and 'Small.
3. *Data type*, whose members represent the allowable types of a prefix and are integer, enumeration, floating point, access, array, fixed point, task, and record.
4. *Size value*, which consists of two sets. One set is the integers within 1 .. P_MAX_SIZE. The other set is the reals within P_MIN_REAL_SIZE .. P_MAX_REAL_SIZE.

The items to be measured are compilation time and size of generated code. The storage allocation mechanism is also assessed.

The state of the compiler is accounted for by the first principal set. The second principal set, *attribute class*, is defined by the language and refers to the attribute designators allowed in a length clause. The *data type* principal set contains all possible types for the prefix of the attributes identified. The last principal set, *size value*, represents the possible values for the expression specified in a length clause. Associated with a length clause is a type declaration, and the value that is specified in the length clause for that type is a function of the range (if present) of the type. That is, the value must be large enough to store the specified range. Not every type identified in *data type* is legal with every attribute identified in *attribute class*, and not every value is legal with every attribute and prefix. Cases such as these must be eliminated from consideration.

3.4. Experimentation for Enumeration Representation Clauses

An enumeration representation clause is defined by Ada as that which allows the user to specify the integer values that correspond to the enumerated literals appearing in the enumeration type. For experimentation we define the principal sets with members as follows:

1. *Compiler state*, whose members represent no optimization, optimization of storage, and optimization of time.
2. *Set size*, which specifies the number of elements in the enumeration representation clause and is in the range 1, 2, ..., P_MAX_SET_SIZE.
3. *Values*, which defines the numeric values that are to be associated with the enumerated literals. The elements of this set are taken from integers in the range P_MIN_VAL to P_MAX_VAL.

The measurables for this set of experiments are the compile time and size of code generated. The manner of storage allocation is also assessed.

The principal set that specifies the values to be associated with the enumerated literal is of special concern in the above description. As indicated above, the values are chosen from integers that lie in some specified range. Performing all possible combinations from possible integer codes is not believed warranted. Rather, the intent is to examine the effects of (a) positive and negative integer codes, (b) noncontiguous integer codes, and (c) any values that are extremely large or small.

3.5. Experimentation for Record Representation Clauses

A record representation clause allows one to completely specify the storage representation of a record so that the order, position, and size of record components can be defined. The following principal sets and associated members are defined for experimentation:

1. *Compiler state*, whose members represent no optimization, optimization of space (storage), and optimization of time.
2. *Record class*, whose members are variant and non-variant.
3. *Discriminant type*, whose members are null, integer, and enumeration.
4. *Number of discriminants*, which specifies the number of discriminants for the record and whose elements are integers in the range 0 .. P_MAX_DISC.
5. *Discriminant order*, whose members are themselves sets that specify the ordering of the possible combinations of elements from the principal set of discriminant types.
6. *Alignment value*, which specifies the alignment position of each record in the subject type and consists of integers in the range 0 .. P_ALIGN_MAX and null.
7. *Component type*, whose members are integer, enumeration, fixed point, floating point, array, access, and record.
8. *Component alignment value*, which specifies the alignment position of a component relative to the start of the record and contains integer elements in the range 0 .. P_CALIGN_MAX.
9. *Component first bit*, which specifies the starting bit of a component and is an integer in the range 0 .. SYSTEM.STORAGE_UNIT - 1.
10. *Component size*, which specifies the length (in bits) of a component and is an integer in the range 0 .. P_MAX_INT.
11. *Number of components*, which specifies the number of components in the record and is an integer in the range 1 .. P_MAX_COMP.
12. *Component order*, whose members are themselves sets that specify the ordering of the possible combinations of elements from the principal set of component types.

The items to be measured include compilation time and size of generated code. An assessment of the storage allocation mechanism is also performed.

The first principal set represents the state of the compiler. The *record class* principal set refers to whether the record is variant or not. The *discriminant type* principal set is identified by the language and consists of the legal types that a discriminant can be. A null element is included in the set to represent the case where there is no discriminant. The fourth principal set gives the number of discriminants. The fifth principal set enumerates all possible orderings of all possible types for the number of discriminants.

The *alignment value* principal set refers to the alignment for each record object of the particular record type. Again, null is an element of this set for the cases when no alignment clause is given.

The seventh principal set, *component type*, is identified by the language and consists of the legal types for a component. Next, the principal set *component alignment value* refers to the system storage unit, starting from the beginning of the record, where the component begins. The *component first bit* principal set represents the values of the beginning bit of a component. The *component size* principal set identifies the possible sizes of a component from which the last bit of a component can

be derived. The next principal set specifies the number of components in the subject record. Finally, the last principal set enumerates all the possible orderings of all possible component types for the number of components.

There is great complexity in experimentation for record representation clauses, as is evident from the above. For example, suppose a record had another record as a component. In this case, the number of possible experiments would be increased *significantly*.

3.6. Experimentation for Address Clauses

Ada provides an address clause that specifies a required address in storage for an entity. For experimentation, the following principal sets and associated members are defined:

1. *Compiler state*, whose members represent no optimization, optimization of space (storage), and optimization of time.
2. *Class type*, which has members variable, constant, subprogram, package, task, and entry.
3. *Address value*, which has members of type SYSTEM.ADDRESS in the range of P_ADDR_MIN_VALUE .. P_ADDR_MAX_VALUE.
4. *Other units*, whose elements are null, procedure, function, package, and task.

The items to be measured include compilation time and the size of generated code. An assessment of the generated code to learn the effects of using an address clause on other program units is of particular interest and could be performed. For example, what happens when an address clause specifies an address that falls within another program unit's allocated storage? When null is chosen from the principal set *other units*, this question is not addressed, by definition.

The first principal set listed represents the state of the compiler. The second set, *class type*, is identified by the language and consists of the entities that may be named in an address clause. The principal set *address value* refers to the possible values of the simple expression in an address clause. The last principal set is included to assess the effect of an address clause in the presence of each member identified.

3.7. Experimentation for Data Conversion and Assignment

There are a variety of techniques through which data conversion and assignment may be accomplished in Ada. Thus, there is the option of performing explicit type conversions where, for example, a value of some integer type is converted to a value of a floating-point type. For the context at hand, we are not concerned with explicit conversions of the type illustrated. We believe that a study of the issues dealing with this area is more properly done in conjunction with experimentation in the area of numerical mathematics.

3.7.1. Use of UNCHECKED_CONVERSION

A basic concern here is with experimentation for the generic function UNCHECKED_CONVERSION. This particular type of conversion may be used to convert a bit pattern from one type to another; note the emphasis on conversion of a bit pattern type, as opposed to a conversion of values. The following principal sets and members are defined below:

1. *Compiler state*, which has members no optimize, optimize time, and optimize space.
2. *Source type*, which has members enumeration, integer, fixed point, floating point, array, and record.
3. *Source size*, which has members representing the number of bits to be allocated for each of the above types. The sizes may range over the integers 1, 2, ..., P_MAX_SOURCE_SIZE.
4. *Source address*, whose elements indicate the effect of data alignment that may be variable with respect to the underlying architecture. The members of this set are themselves two sets. The first set has elements that range over the integers 1, 2, ..., P_MAX_STORAGE, where the upper bound specifies the maximum multiple of system storage unit to be considered. The second set contains elements 1, 2, ..., P_MAX_BIT_STORAGE, where the upper bound specifies the largest bit compatible with an integral multiple of the system storage unit. The members of this set may be seen to be related to an aligned record representation.
5. *Target type*, whose elements are the same as the *source type*.
6. *Target size*, whose elements are the same as those of *source size*.
7. *Target address*, whose elements are the same as those of *source address*.

The items to be measured for the above include compilation, execution times, and generated code size. The manner by which the conversion is accomplished is also studied. For example, a compiler could affect the conversion by using a call to the runtime library.

In the above experiment considerable effort is taken to examine the relative data locations with respect to the machine architecture. That is, the manner in which the conversion is accomplished may depend on whether the data are aligned modulo the system storage unit. It is this latter point that is examined by the use of the *source size* and *source address* sets, for example.

3.7.2. Data Assignment

The relevant principal sets and their associated members may be inferred from the above discussion. We must include an additional principal set to account for the effect of pragma PACK. For this case the principal sets are:

1. *Allocation class*, whose members represent packed and not packed.
2. The principal sets defined in Section 3.7.1.

Here, the specific values may be obtained by requiring applicable constraints. For example, it is required that the source and target types be identical. The question of execution time for accessing data within a particular structure is within the scope of data assignment.

3.8. Experimentation for Representation Attributes

Representation attributes are provided by Ada that allow a user to obtain characteristic machine-dependent values. The experimentation in this general area is divided into three categories, discussed below.

3.8.1. Experimentation for X'ADDRESS

One representation attribute, X'ADDRESS, may be used to obtain information about addresses where an item X is allocated. In this case, the relevant principal sets and their members are:

1. *Compiler state*, whose members represent no optimization, optimization of space (storage), and optimization of time.
2. *Class*, whose members are two sets. The first set, Object_Set, contains members representing enumeration, integer, fixed point, floating point, record, and array type. The second set, Unit_Set, contains members representing subprogram, package, task, label, generic, and entry.

The principal measures for this experiment are the code size and characteristic execution time for the generated code to determine the value of X'ADDRESS for a member of the *class* principal set. The manner in which the value of 'ADDRESS is determined by the code is assessed.

3.8.2. Experimentation for X'SIZE

A second form of representation attribute defined in Ada allows one to obtain information about storage allocation. In this case there are four principal sets, in particular:

1. *Compiler state*, whose members represent no optimization, optimization of space (storage), and optimization of time.
2. *Data type*, which contains members representing enumeration, integer, fixed point, floating point, and record types.
3. *Type size*, which is used to account for the presence of a representation clause specifying the size of the data types considered. This set has members null and integers in the range 1 .. P_MAX_SIZE. The value of null reflects the option of not using a length clause with the SIZE attribute designator.
4. *Data range*, which contains members P_MIN_VAL and P_MAX_VAL that are used to complete the definition of the numeric types indicated above. Note that the values of these quantities are a function of the amount of storage to allocate if a length clause with the SIZE attribute designator is present.

For this case, the measurables are the amount of code generated and characteristic execution time. The emphasis here is on assessing the manner in which the code is generated to provide a value for X'SIZE.

3.8.3. Experimentation for Record Types

The final type of representation attribute allows one to obtain information about the location of a component C of some record R. The location of a component may be determined by using R.C'POS, which provides the position of the specified record component. Additionally, the representation attributes R.C'FIRST (R.C'LAST) may be used to determine the first (last) bit address of the component. In this case the principal sets are:

1. *Attribute type*, which contains members representing 'POS, 'FIRST, and 'LAST.
2. The principal sets defined in Section 3.5.

The experimentation in this area is designed to measure the amount of code generated and characteristic execution time for use of the above representation attributes. The manner in which code is generated for the above representation attributes may be assessed.

Note the necessity of including the principal sets defined in connection with experimentation for record representation clauses, discussed in Section 3.5. In the present case of representation attributes, they return values for an attribute of a record component. For this reason, a record must be defined as part of the experimental setup; this is reflected by a reference to the principal sets used in the experimentation for record representation clauses. Stated differently, to test the code generated for the application of the representation attributes, a record must be present for these attributes to operate on. The increased complexity of the associated experimentation is implicit in the above remarks.

4. Methodology

The principal goal of this report is the description of a general methodology applicable to the assessment of support for representation clauses and implementation-dependent features by a particular Ada compiler. The discussion of the preceding chapters has set the stage for presentation of a methodology. In Chapter 2, we outlined the type of investigations possible, ranging from case studies to detailed experimentation. The latter subject was examined at length in Chapter 3. We now build on this foundation and examine the question of methodology development, which is applicable to the present discussion. While the discussion to follow is directed at representation clauses and implementation-dependent features in particular, it is important to note that the methodology is of sufficient generality that it may be applied to other areas of compiler assessment.

4.1. Motivation

We have noted that the emphasis here is on the development of a methodological framework that can be applied to the testing of compiler support for representation clauses and implementation-dependent features in Ada. The principal line of development is the use of detailed experimentation, as discussed in the preceding chapter. Such detailed experiments are considered here for several reasons, among them:

1. They are typically resource-intensive, both in terms of machine allocation and labor.
2. They provide the opportunity for the collection of a considerable body of data containing a wealth of information.
3. They present challenges to the development of effective techniques, not only for testing, but for analysis of the collected data.

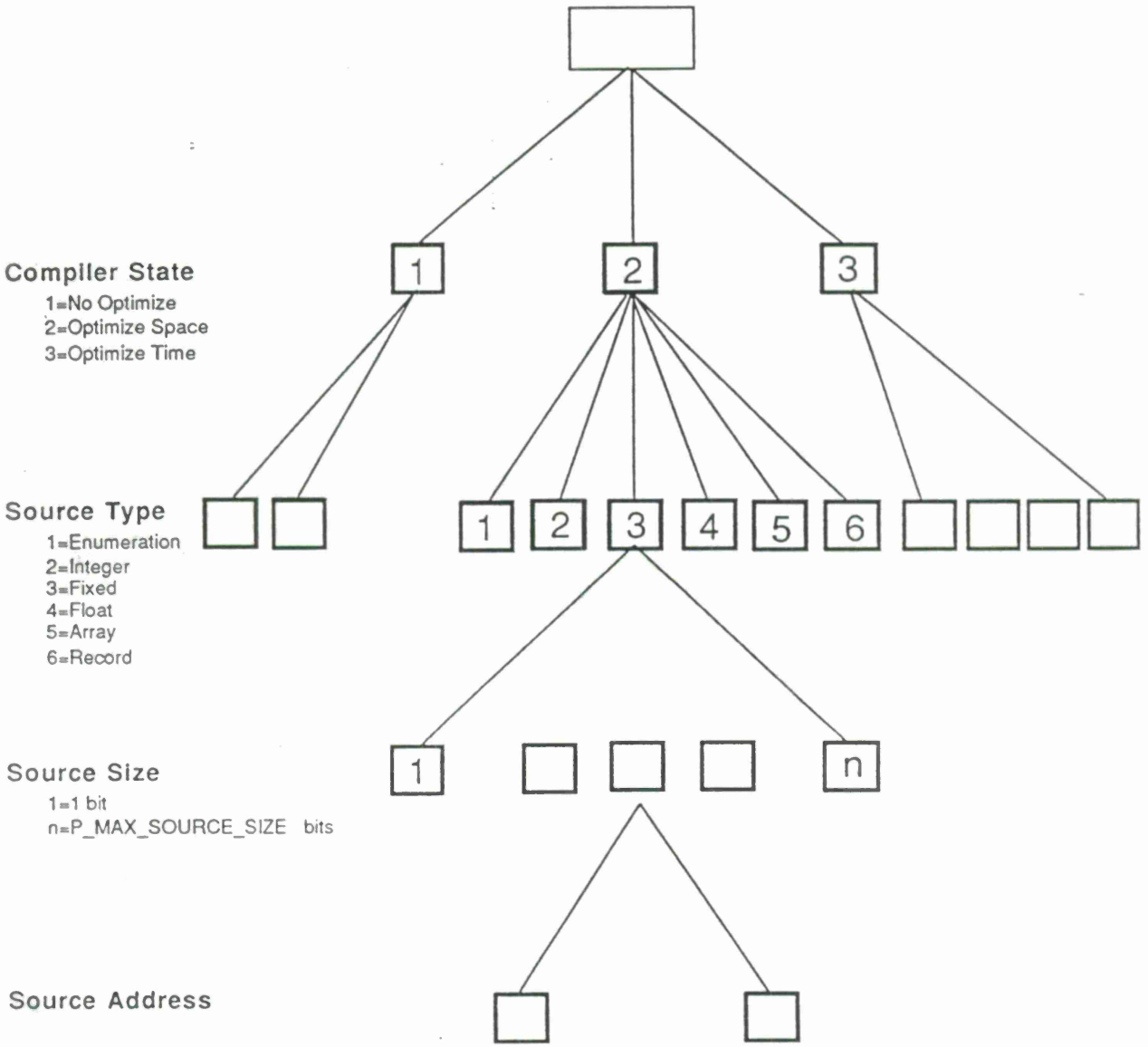
Thus, detailed experiments, as indicated above, present contrasting perspectives. On the one hand, they typically are resource-intensive, requiring considerable effort. On the other hand, they are capable of providing a potential wealth of information. A basic goal of the methodology to be developed is to address these challenges.

It is worthwhile to note that the principal sets and domain of the basic variables, as considered in the preceding chapter, may be examined in the context of a tree structure. The root of the tree is a manifestation of the problem to which the experiment is addressed. Each level of the tree represents consideration of one of the principal sets.

Figure 2 presents an example of the tree structure for experimentation relating to UNCHECKED_CONVERSION. The design methodology for this case was discussed in Section 3.7.1. The first level of the tree represents the compiler state, specifically, no optimization and optimization of time and space. For each node in the first level of the tree there are six successors. These represent the elements of the principal set *source type* and may be enumeration, integer, fixed point, floating point, array, and record. For each of these nodes there are successors representing the principal set *source address*. The remainder of the (partial) tree structure shown in Figure 2 may be enumerated according to the discussion in Section 3.7.1.

Figure 2

Partial Tree Structure for UNCHECKED_CONVERSION



Discussing the application of detailed experimentation in terms of a tree structure illustrates several important points:

1. Typically, the number of nodes and paths in the tree will be quite large, especially if one is interested in extensive coverage.
2. A case study, where a member of any principal set(s) is chosen for the case study, is equivalent to a path through the tree.
3. The fact that the number of paths in the tree is large lends importance to the ability to perform pruning of the tree.

When phrased in the above context, it is clear that there are several major issues that need to be confronted by any methodology that pretends to offer completeness in the problem domain. This, and some justification for the technique, are provided below.

4.2. Overview of the Procedure

The development of a methodology that is capable of the type of assessments indicated above presents a considerable challenge. First, the methodology must minimize the resource-intensive nature of the process. This implies a second point, namely, the method must make use of automated tools wherever possible. Finally, the overall framework should make use of constraints in the application of experiments.

The above points are illustrated in the methodology. In the following, we present a high-level discussion of a methodology based on detailed experimentation.

1. Formulate the problem to be addressed.
2. Identify the quantities that are to be measured for the experiment.
3. Identify the items that are to be assessed, though perhaps not in quantitative terms.
4. Identify the principal sets that are applicable to the scope of the problem.
5. Identify the elements of the respective principal sets that are applicable to the scope of the problem.
6. Identify the constraints that are applicable to the scope of the problem.
7. Create program generators to automate the conduct of the experiment, including the collection of required data.
8. Create analyzer programs to analyze the data collected.
9. Based on the output of the analysis tools, determine the conclusions from the experiment.

The above is an overview of the general procedure deemed applicable to the use of detailed experiments. As noted earlier, our interest is with the application of the above methodology to compiler support for representation clauses and implementation-dependent features in Ada. However, the above discussion is sufficiently general that it may be applied in other settings.

The first step is to define the nature and scope of the experiment. It is this step that must be addressed with care for it affects all that follows. A part of this effort should include consideration for the goals of the experiment and the manner in which the results will be used. A natural outcome of this first step is the identification of what the expected output from the experiment will be. Hence, in

step two, an identification of the quantifiable measures of the experiment are to be defined. This is followed, in step three, by a definition of the nonquantifiable items that are desired from the experiment.

Given that one has identified what is to be assessed, the next step in the procedure is to examine the principal sets that are required for consideration. In this sense, one may think of the principal sets as the primitives that are applied to construct the experimental domain for any experiment. Having identified the relevant principal sets, in step five the elements of the principal sets that are relevant to the experiment at hand are defined.

Steps one through five result in a concise definition of the scope of the problem to be addressed. Moreover, the experimental domain has also been defined with the identification of relevant principal sets and members of the sets that are to be included in the construction of the experiment. Upon completion of step five, it is possible to enumerate the tree structure of the chosen experiment.

In step six, it is requested that the constraints on the experimental domain be identified. An example of how this may be accomplished is through the use of a qualitative assessment of the chosen compiler. In reference [4], we have provided a set of questions relevant to the assessment of a given compiler regarding representation clauses and implementation-dependent features. It is through the use of the results obtained from a qualitative examination of the compiler by which one may begin to prune the tree. An example of how this may be accomplished is provided below. However, the importance of this step cannot be overlooked; the ability to prune the tree is important for it will indicate the amount of effort required to complete the experiment.

Steps seven and eight involve the use of program generators to actually perform the experiment, including data collection and the application of analysis tools, respectively. These topics are of special importance; for this reason, they are discussed independently in the following sections.

The last step in the methodology described is to assess the results obtained. This may involve consideration of the ability to draw general conclusions, and perhaps to suggest other experimental approaches to additional, associated problems. It is in the last step – and only in the last step – that support is generally provided for case studies. In other words, the justification of a case study requires qualification. Only after conducting detailed experiments can this justification be accomplished.

4.3. The Use of Program Generators

It has been noted several times that there is potential for collecting large quantities of data. Correspondingly, there may be a need for developing a large number of test programs. These two points imply the need for some automated procedure to conduct the experiment and to collect the data.

One of the basic steps in the methodology outlined above is to apply program generators to automate the performance of an experiment and data collection. Basically, a program generator is a program that generates one (or more) programs from a set of specifications. The application of program generators has been used by one of us (BCM) with considerable success in the analysis of large

large datasets. The technique of using generators applies equally well to the domain at hand. In fact, the use of such techniques is almost a necessity to cover the range of experimentation suggested in this document.

Recognizing that a program generator is a program, or tool, that generates other programs, it is well to illustrate the methodology associated with this technology. In the area of experimentation for compiler support of representation clauses and implementation-dependent features, a generator would typically include the following steps:

1. Generate an Ada source program for the appropriate experiment.
2. Compile and link the generated program.
3. Execute the generated program, if necessary.
4. Invoke an analysis tool to collect information.
5. Invoke an analysis tool to examine the collected information.

The first step in the use of a generator is for some program to create a second program (in Ada). This generated program is written according to some specification. For example, the specification might be to generate a program using particular principal sets and members of the principal sets. Typically, part of the generated program will be constant, while other parts will vary.

The second step is to automatically compile and link the created program. This may be accomplished by spawning a process.

The next step involves executing the created program. Note that this step is optional. If the goal of an experiment stopped at the analysis of generated (assembler) code, it would be unnecessary to execute the generated code.

In step four of the procedure, an analysis tool is invoked to collect some information. The information collected may be considered data to be used for other analysis tools. Similarly, the data collected may not require any further (automated) analysis.

The last step in the procedure is to invoke an analysis tool. The input to this tool would be the information collected in the previous step. At this stage in the methodology, one typically is performing some type of "global" analysis on the data collected. It is from this and the preceding step that the results of the experiment are provided.

It is especially important to note that all of the above steps may be accomplished under control of one program. Thus, this one program generates and compiles a second program. It may be this same program that invokes all tools necessary to collect and analyze any relevant data. In fact, in some instances, the control program may itself be generated, which in turn creates other programs.

To illustrate the above techniques, a simple example of using a program generator is presented in Section 4.5 below.

4.4. The Use of Analysis Tools

Associated with the need to automate the creation of programs, there is a corresponding need to automate the analysis of the data collected. If the data analysis phase of the application of program generators cannot be automated, there is a clear drawback to the use of the method. Hence, in the following, some mention of analysis tools is made. The discussion is illustrative in nature. A complete discussion of the creation and use of analysis tools is a broad topic in itself. However, enough information is provided to indicate the nature of the problem.

At two places in the preceding discussion, mention was made of the use of analysis tools. The application of such tools may be divided into two basic categories: tools that collect information about some particular control variable, and tools that perform analysis on the collected data. As indicated above, the latter are typically global type analysis tools that are invoked after all the data have been collected.

We begin our discussion of analysis tools by considering those tools that may be used for collecting data. The model for the tool is that there exists a well-defined input and output to the tool. Although this appears a trivial statement, it is well to consider in the design of tools for the purposes envisioned here. An additional assumption, also affecting the tool design, is that we assume there exists a one-to-one relationship between what the tool collects and the problem domain. In other words, each tool collects one unique piece of information. This assumption is made in the interest of simplicity. From the perspective of tool development, the preceding implies a set of tools that address the data primitives to be collected.

Data collection tools may vary considerably in scope. That is, the information collected may vary both in quantity and complexity. For purposes of illustration, a representative list of tool primitives would extract the following information:

1. *Code size:* Input to the tool is the program listing; from this the tool extracts the size of the generated object code.
2. *Compile time:* Input to the tool is the program listing; from this the tool extracts the compile time. For those Ada compilers that provide compilation time by phase (such as parse, optimization, etc.), this tool can be refined accordingly.
3. *Recorded execution time:* Input to the tool is the program listing; the tool extracts the execution time.
4. *Instruction set frequency:* Input to the tool is the assembler code from the compiler; the tool provides a frequency count of the instruction set usage. This tool can be extended to include macro statement references by the compiler as well.
5. *Runtime references:* Input to the tool is the assembler code from the compiler; the tool provides the number of calls generated to the runtime library. The tool may also provide which runtime library routines are referenced.
6. *Branch displacement counts:* Input to the tool is the assembler code from the compiler; the tool provides information about the branches generated by the compiler. Of relevance here is the "distance" from the source and target of the branch. In particular, if one is assessing a virtual machine, one may be especially interested in the number of branches that are to addresses on another page of memory.

7. *Instruction length distribution:* Input to the tool is the assembler code from the compiler; the tool provides a frequency distribution of the instruction lengths used in the code. Thus, a characteristic feature of instruction sets is that they provide instructions of different lengths, and one may be interested in the relative frequency of the instruction lengths. Such information may be useful, for example, in the analysis of different optimization criteria.
8. *Execution emulation:* Input to the tool is the assembler code from the compiler; the tool emulates the generated code. A tool such as this is typically complex, though capable of providing a wealth of information. Where especially useful, such a tool provides execution-timing behavior that may be traced back to the source program. In this case, the output can be used to determine potential bottlenecks in the source code.

The above list could continue with the addition of many other tools. A characteristic that appears from the preceding development is that the creation of tools for the type of analysis being considered may be a nontrivial task. That is, if one wishes to examine several different compilers and a detailed amount of information is desired, then a large tool set is required. In particular, one may typically have a tool for a given primitive for a given compiler.

The second type of tool applicable to the assessment of compilers is an analysis tool. This particular type of tool would take as input the output from one or more data collection tools listed above. Based on information collected by the other tools, it is the purpose of this second type of tool to perform some type of "global" analysis. This may include, for example, statistical computations for input data. Thus, one may be interested in the average compilation time or a plot of the code generated for each source statement. A myriad of possibilities exists here, all of which are intended to process a possibly large dataset and search for meaningful conclusions.

The type of global analysis tools considered here may quite well apply techniques from artificial intelligence. For example, it is possible to construct an expert system that applies rules to assess the collected data. The application of expert systems to assessments such as this is an area worthy of investigation.

The development of global analysis tools, as indicated above, is independent of a particular compiler. That is, one may presume the existence of primitive data collection tools that are compiler-specific. The output from these tools is used by an analysis tool that is largely independent of any compiler. Hence, regarding the global type analysis tool, there is the possibility of reuse of the tools.

4.5. An Example of the Procedure

We now consider an example of the use of the above procedures. In particular, we are especially concerned with the application of program generator techniques.

For the domain of the example, we assume that an application program needs to perform a considerable amount of data conversions from an integer representation to a fixed point representation. We also assume that the application is interested in applying the use of the generic function UNCHECKED_CONVERSION. Experimentation for this function was discussed in Section 3.7.1. A partial tree structure for experimentation relevant to this function was presented in Figure 2.

For the formulation of the problem, we assume that the application has integer (source) data whose size varies from one to 16 bits. It is also assumed that the size of the target fixed-point type will vary. In particular, we assume that the 'SMALL attribute will range over integral powers of two with the exponent ranging from 1 to 16.

The above represents a partial definition of the problem. To continue, we assume that it is desired to measure the code generated and a characteristic execution time. For these items, we shall include the effect of compiler optimization. That is, we include the pragma OPTIMIZE and study optimization of time and space. We assess the manner in which the UNCHECKED_CONVERSION is accomplished. We do not force either the source or target type to be aligned on a specific boundary; rather, the data assignment is left to the compiler. We also assume that it is desired that the target size vary from 16 to 32 bits.

Upon completion of the above, the first three steps of the methodology discussed in Section 4.2 are defined. That is, we have formulated the problem and determined what is to be measured and assessed. The next step in the methodology is to determine the principal sets and the appropriate members of the sets. For the problem at hand, this was accomplished as part of the formulation of the problem.

The next step in the approach is to examine the applicable constraints to the (proposed) experiment. Information relative to the constraints is available from qualitative assessments of the compiler, for example. We assume that a qualitative evaluation of the compiler has been completed. Thus, one of the evaluation questions listed in reference [4] refers to the representation of fixed-point data types. We assume that the compiler in question only provides for a fixed-point data type that is represented as 32 bits. Recognition of this is equivalent to identifying an applicable constraint. This means that it is not necessary to vary the size of the (target) fixed-point data type from 16 to 32 bits as originally envisioned. Rather, the constraint that fixed-point types must be represented as 32-bit quantities is a limitation on the permitted members of the principal set dealing with the target size. The recognition of this constraint decreases the dimensionality of the problem by a factor of 16.

Mention has been made of the dimensionality of the problem concerning the application of a recognized constraint. For this example problem, what is the dimensionality? We have considered two states of optimization, 16 values of source size, and 16 values of the 'SMALL attribute for the target size. Hence, the dimensionality of the experiment is 512. In other words, the experiment under consideration comprises 512 separate tests. Stated differently, it is required that 512 programs be written, data collected and the results analyzed.

In light of the number of programs required, there is considerable motivation for the application of program generators. For the case at hand, the program generator is fairly simple. To consider the question of a program generator for this example, we develop a skeleton program to implement the desired experiment. From the skeleton program we then construct the applicable generator to invoke data collection and analysis tools.

A skeleton Ada program for the example is shown in Figure 3. There, elements shown in boldface prefixed by an ampersand are related to the principal sets and the values of the elements of those sets. The items in boldface may be thought of as slots to be filled in by the generator. There are slots for the compiler optimization state, the size of the source type, and the values of the target type delta. Note also that it is required to have values for the ranges of the types referenced. These ranges are to be computed as a function of the size of the respective fields.

Based on the skeleton code presented in Figure 3, we now construct the appropriate program generator. A pseudo-Ada PDL for the generator is shown in Figure 4. The upper part of the PDL performs a loop over the three principal sets, namely, compiler state, source size, and delta. A part of this loop also performs the calculations for the ranges of the latter quantities.

Figure 3

Skeleton Code for Simple Unchecked_Conversion Experiment

```
with Unchecked_Conversion;
procedure Convert is

   pragma Optimize (&STATE);

   type Int is range -(Int_Last + 1) .. Int_Last;
   for Int'Size use &SIZE;

   type Fixed is delta &DEL range -(Fixed_Last + 1) .. Fixed_Last;

   function Convert_to_Fixed is new Unchecked_Conversion (Int, Fixed);

   I : Int;
   F : Fixed;

begin -- Convert

   F := Convert_to_Fixed (I);

end Convert;
```

Figure 4

Pseudo-Ada PDL for Program Generator for Simple Unchecked_Conversion Experiment

```
procedure Generator is

  type Compile_State is (Time, Space);

  type Source_Type is range 1 .. 16;

  type Delta_Type is range 1 .. 16;

begin

  for I in Compiler_State loop

    for J in Source_Type'Range loop

      for K in Delta_Type'Range loop

        &STATE = I;
        &SIZE = J;
        &DEL = 2**(-K);

      -- Compute range of source and target types using &SIZE and
      -- &DEL.

      -- Generate code from skeleton in Figure 3 using values of
      -- &STATE, &SIZE and &DEL.

      -- Compile code; send source output to a file C.OUT; send assembler
      -- listing to a file A.OUT.

      -- Invoke a data collection tool that takes as input the file
      -- C.OUT and returns the size of the source code (in bytes) as
      -- Code_Size.

      -- Invoke a data collection tool that takes as input the file
      -- C.OUT and returns the compilation time as Compile_Time.

      -- Invoke a data collection tool that takes as input the file
      -- C.OUT and performs a frequency count of instruction set usage
      -- associated with the source statement 'F  :=  Convert_to_Fixed(I)'.
      -- The output from this tool is a two-dimensional array indicating
      -- the assembler statement and frequency of use.
```

Figure 4 (Continued)

```
        -- Invoke a data collection tool that takes as input the output
        -- of the tool referenced above.  This tool computes code size for
        -- the 'F  : = Convert_to_Fixed(I)' statement.  The tool also returns
        -- a characteristic execution time based on frequency counts of
        -- instructions referenced.

        -- Invoke a tool that takes as input the file A.OUT and counts the
        -- number of runtime library calls associated with the statement
        -- 'F  : = Convert_to_Fixed(I)'.

        -- Record:
        --   State variables (&STATE, &SIZE, &DEL)
        --   Total source code size (Code_Size)
        --   Total compile time (Compile_Time)
        --   Most frequently used instruction and number of references
        --   Code size for conversion statement
        --   Characteristic execution time for conversion statement
        --   References to runtime library

      end loop;

    end loop;

  end loop;

  -- Invoke a formatter tool that takes as input the recorded data
  -- and pretty prints the results.

end Generator;
```

The program generator shown in Figure 4 closely follows the outline discussed in Section 4.4. Thus, after the code is generated, it is compiled, and the output from the compilation is sent to a file. Several data collection tools are referenced in Figure 4. The first two tools extract the size of the source code and the compilation time.

The second two tools referenced in Figure 4 are somewhat more complex than the simple tools noted above. Thus, we first invoke a tool that performs a frequency count of the instruction set used. Note the indicated limitation on how this tool is used. We only perform a frequency count for instructions generated from the use of the assignment statement that invokes a function that is an instantiation of UNCHECKED_CONVERSION in the source code. Following this, another tool is invoked that provides a characteristic execution time for the statement invoking the function. The characteristic execution time is obtained by using the instruction set frequency distribution and representative execution time for the associated instructions. Finally, another tool is invoked that determines if the unchecked conversion is accomplished by a call to a runtime library. At the bottom of the inner loop another tool is invoked that records the collected information.

At the conclusion of the loops over the members of the principal sets, 512 programs are generated and compiled. Additionally, each of these programs will have been operated on by data collection tools that provide the relevant measures for this experiment. This information is recorded for use by later stages in the analysis. It is at this latter point that global-type analysis tools can be invoked.

In the present example, one possible global analysis tool would be simply a formatter for the collected data. Such a formatter could report the optimization state, source field size, value of target field delta, compile time, characteristic execution time, code size, instruction set frequency counts, and calls to the runtime library. This collected information would then be available for examination. The ability to present such data in a collective manner is possible only after executing the 512 programs and invoking a tool to extract each desired piece of information.

To complete the discussion, the last step in the methodology discussed in Section 4.2 involves the development of conclusions. For the problem we are considering here, two of the relevant questions that may lead to the formulation of conclusions are:

1. How does the size of the generated code and the average execution time depend on the state of optimization?
2. How does the instruction set frequency usage depend on the state of optimization?

There are additional questions that could be added to the above list, although we will not pursue this path. Rather, we believe it is worthwhile to make some general remarks concerning the above example. First, it should be apparent that one must exercise caution when comparing generated code for different optimization states; occasionally, certain code segments may be optimized away, for example. A second point is that despite the simplicity of the example considered, it clearly illustrates the motivation for the use of program generators and automated analysis tools. Another point is that this example could be considerably extended. Thus, we could have also allowed for the source (or target) to be a component of a record. Lastly – and this must not be overlooked – automating the generation and data collection provides a wealth of material; a major challenge is extracting meaning from such a large amount of available data.

4.6. Summary

This chapter has discussed a methodology applicable to compiler assessment. The domain has been for assessment of support provided for representation clauses and implementation-dependent features, although the methodology is considerably more general. The emphasis has been on application of detailed experimentation as this provides comprehensive information and presents significant challenges. The unique elements of the methodology are the use of program generators and automated analysis tools. An example was considered in light of the methodology. In summary, we believe that the methodology outlined above has considerable application to the problem area under discussion, as well as to other problem domains.

5. Summary

This report is one in a series dealing with the use of representation clauses and implementation-dependent features in Ada. The intent of the present document is to define and illustrate experimental procedures for the assessment of support provided for representation clauses and implementation-dependent features by a particular compiler. This is warranted by the following two considerations:

1. Representation clauses and implementation-dependent features are expected to find application to many mission-critical systems.
2. The support provided for representation clauses and implementation-dependent features is expected to vary among different compilers.

The principal focus of this document has been to develop a methodology applicable to the assessment of support provided by a given compiler for representation clauses and implementation-dependent features. The emphasis here has leaned toward detailed experimentation, as opposed to selected case studies. The use of detailed experimentation is expected to be both challenging and resource-intensive. To address these factors, the methodology incorporates program generators. That is, a fundamental element of the methodology developed here is to apply programs that create other programs and invoke automatic analysis tools. An example of the procedure was treated in some detail.

References

1. DoD Instruction 5000.31, July 1986. DoD Directive 3405.2, March 30, 1987.
2. *Reference Manual for the Ada Programming Language*, Department of Defense MIL-STD-1815, 1983.
3. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada. I: Overview*, CMU/SEI-87-TR-14, ESD-TR-87-115, July 1987.
4. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada. IIA. Evaluation Questions*, CMU/SEI-87-TR-15, ESD-TR-87-116, July 1987.
5. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada. IIIA: Qualitative Results for VAX Ada Version 1.3*, CMU/SEI-87-TR-17, ESD-TR-87-118, July 1987.
6. B. Craig Meyers and Andrea L. Cappellini, *The Use of Representation Clauses and Implementation-Dependent Features in Ada. IVA. Qualitative Results for Ada/M(44), Version 1.6.* CMU/SEI-87-TR-19, ESD-TR-87-170, July 1987.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-87-TR-18	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-87- 125
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) The Use of Representation Clauses and Implementation-Dependent Features in Ada: IIB. Experimental Procedures	
12. PERSONAL AUTHOR(S) B. Craig Meyers and Andrea L. Cappellini	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) July 1987 15. PAGE COUNT 31
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) representation clauses in Ada implementation-dependent features in Ada experimental procedures

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report is one in a series dealing with the use of representation clauses and implementation-dependent features in Ada. The purpose of this report is to discuss detailed experimental procedures to assess compiler support. It is readily acknowledged that the domain of possible experimentation is large. To facilitate the experimentation, a methodology is proposed that relies on program generators and automated analysis tools. An example of the methodology is presented in some detail.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED XX SAME AS RPT. ☐ DTIC USERS XX	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473 83 APR EDITION OF 1 JAN 73 IS OBSOLETE